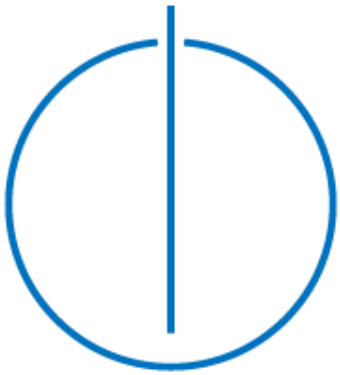
Technical University of Munich
Department of Informatics

Master's Thesis in Informatics

Deep Learning for Protein Function Prediction

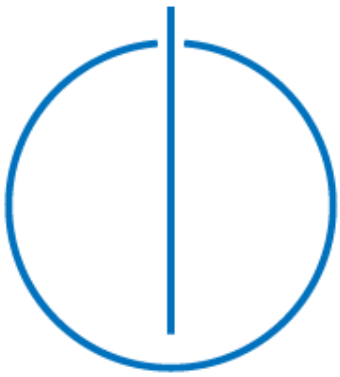
Manuel Alba Avilés



Technical University of Munich
Department of Informatics

Master's Thesis in Informatics

Deep Learning for Protein Function Prediction
Tiefe künstliche neuronal Netze für Vorhersage von Proteinfunktion



Author: Manuel Alba Avilés
Supervisor: Prof. Dr. Daniel Cremers
Advisor: M.Sc. Vladimir Golkov
Submission: September 26, 2017

I confirm that this master's Thesis is my own work and I have documented all sources and material used.

Munich, September 26, 2017

(Manuel Alba Avilés)

Abstract

The assignment of functions to proteins is a bottleneck due to the need of costly and time-consuming molecular experiments. This is the reason why more often data analysis methods are used for protein annotation. In this thesis I consider an approach based on Deep Learning architectures to automatically annotate proteins. A model that uses Convolutional Neural Networks was implemented using position-specific score matrices built from protein clusters as input. Although suitable architechtures were found, the data sparsity, represented by the amount of possible annotations for a protein taken into account, was a too much complicated problem to overcome. Therefore, to face this situation, data augmentation in the form of phylogenetic tree pruning of the Multiple Sequence Alignments was used, improving the prediction quality of the model but still not being able to provide accurate predictions. Direct comparison of this new approach with Critical Assesment of Functional Annotation methods is difficult due to the need of working with precomputed data because the time-consuming preprocessing required to obtain the input data. Position-specific score matrices provide an insight on the protein functions although more information/data is needed in order to correctly predict all the protein annotations desired.

Acknowledgements

First I would like to express my sincere gratitude to my advisor Vladimir Golkov for the continuous support of my Master Thesis, his insightful comments and encouragement and the chance of doing my project with him in first place. Besides of my advisor, I would like to thank Anne Rickmann, Tomislav Tomov and Sebastian Bachern, who worked on related projects and helped me with countless ideas, different points of view and good insights, becoming a great support team for the project. Finally I want to thank Quirin Lohr for his support regarding the computers on which the networks were trained and all data was stored, and the computer vision chair for providing the necessary hardware.

Contents

List of Figures

List of Tables

1 Introduction

1.1 The importance of proteins in life

Proteins are molecules that are made of amino acids. They are stablished by our genes and form the basis of living tissues. They also play a fundamental role in biological processes. For instance, proteins catalyze reactions in our bodies, transport molecules such as oxygen, keep us healthy as part of the immune system and transmit messages from cell to cell. Proteins are the building blocks of life and come in many different shapes and sizes. Each protein has a specific role or function in our body, and some of them even perform more than 1. Here are some examples of different protein functions:

- Enzymes: facilitate biochemical reactions. An example is pepsin, which is a digestive enzyme in your stomach that extract proteins from food.
- Antibodies: produced by the immune system to fight infections and remove foreign substances.
- DNA-associated proteins: regulate chromosome structure during cell division and/or help to regulate gene expression. Examples are histones and cohesion.
- Contractile proteins: involved in muscle movement and contraction. Examples are histones and cohesin.
- Structural proteins: provide support in our bodies. Examples are collagen and elastin, which belong to our connective tissues.
- Hormone proteins: co-ordinate bodily functions. An example is insulin, which controls the sugar concentration in our blood by regulating the uptake of glucose into the cells.
- Transport proteins: move molecules around our bodies. An example is haemoglobin, which transports oxygen through the blood.

In consequence, proteins are really important to understand life and how humans interact with it.

1.2 Critical Assesment of Functional Annotation (CAFA)

The project is based on CAFA2 paper, based on identifying protein functions using the information provided: protein sequences and Gene Ontology (GO or annotation). The GO project provides controlled vocabularies of defined terms representing gene product properties. These cover three domains: Cellular Component, the parts of a cell or its extracellular environment; Molecular Function, the elemental activities of a gene product at the molecular level, such as binding or challenge

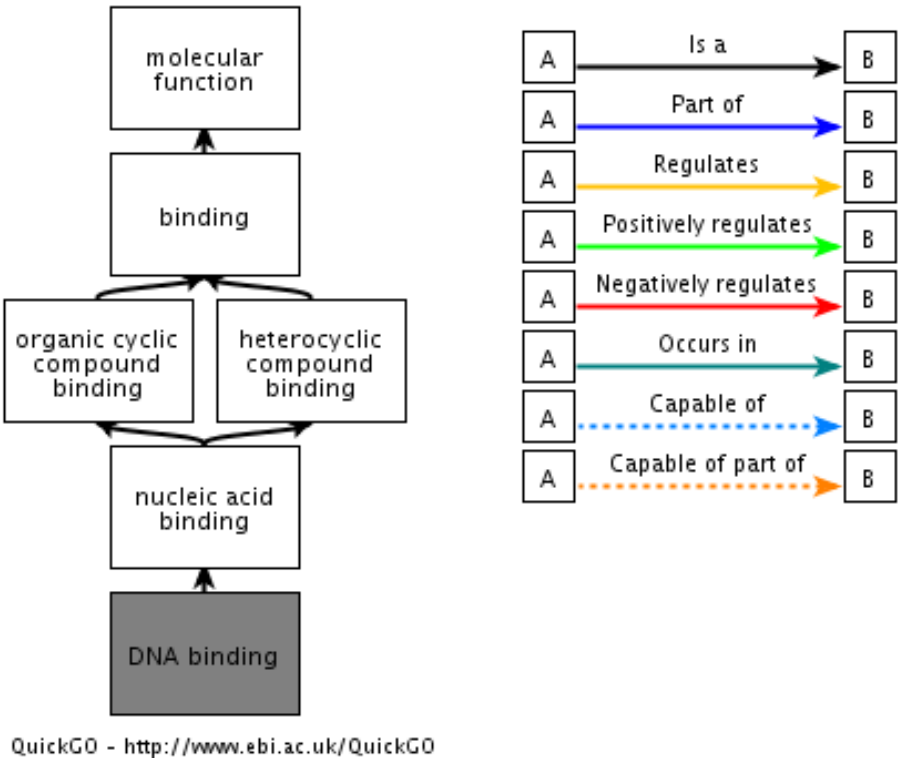


Figure 1: Example of GO:0003677 (DNA binding)
Extracted from: `http://amigo.geneontology.org/amigo/term/GO:0003677`

The GO vocabulary is designed to be species-agnostic, and includes terms applicable to prokaryotes and eukaryotes, and single and multicellular organisms. In an example of GO annotation, the gene product cytochrome c can be described by the Molecular Function term oxidoreductase activity, the Biological Process terms oxidative phosphorylation and induction of cell death, and the Cellular Component terms mitochondrial matrix and mitochondrial inner membrane.
The annotations used could be of the following types:

- Biological Process Ontology (BPO): biological process term describes a series of events accomplished by one or more organized assemblies of molecular functions. Examples of broad biological process terms are cellular physiological process or signal transduction. Examples of more specific terms are pyrimidine metabolic process or alpha-glucoside transport. The general rule to assist in distinguishing between a biological process and a molecular function is that a process must have more than one distinct steps. A biological process is not equivalent to a pathway. At present, the GO does not try to represent the dynamics or dependencies that would

be required to fully describe a pathway.

- Cellular Component Ontology (CCO): These terms describe a component of a cell that is part of a larger object, such as an anatomical structure (e.g. rough endoplasmic reticulum or nucleus) or a gene product group (e.g. ribosome, proteasome or a protein dimer).
- Molecular Function Ontology (MFO): Molecular function terms describes activities that occur at the molecular level, such as catalytic activity or binding activity. GO molecular function terms represent activities rather than the entities (molecules or complexes) that perform the actions, and do not specify where, when, or in what context the action takes place. Molecular functions generally correspond to activities that can be performed by individual gene products, but some activities are performed by assembled complexes of gene products. Examples of broad functional terms are catalytic activity and transporter activity; examples of narrower functional terms are adenylate cyclase activity or Toll receptor binding. It is easy to confuse a gene product name with its molecular function; for that reason GO molecular functions are often appended with the word activity.

In CAFA2 challenge, the main idea is using the **protein sequences** as an input for building a predictive model that accurate gives the protein annotations. Although other information could be really useful for this task, such as **protein structure**, it is presented in very few proteins and therefore the analysis is focused on the sequences.

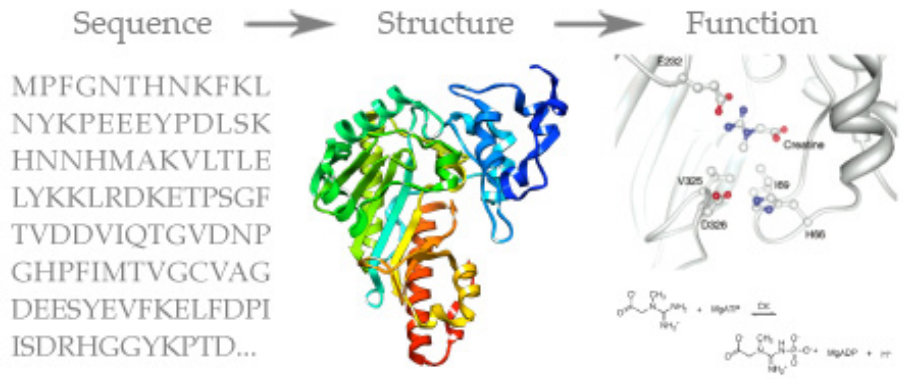


Figure 2: Example of Protein Sequence and Structure
Extracted from: `http://persweb.wabash.edu/facstaff/novakw/`

CAFA2 challenge was organized in 3 stages:

- t_{-1}: the organizers provide the protein sequences that will be used in the prediction challenge. Teams can use them in order to develop the models designed for annotating the proteins.

- t_0: the prediction phased ends and the prediction models are submitted. New sequences that now have experimental annotation are collected to use them as benchmark.
- t_1: evaluation. The prediction methods presented are tested against the sequences collected in t_0.

Here it is presented the structure of CAFA2:

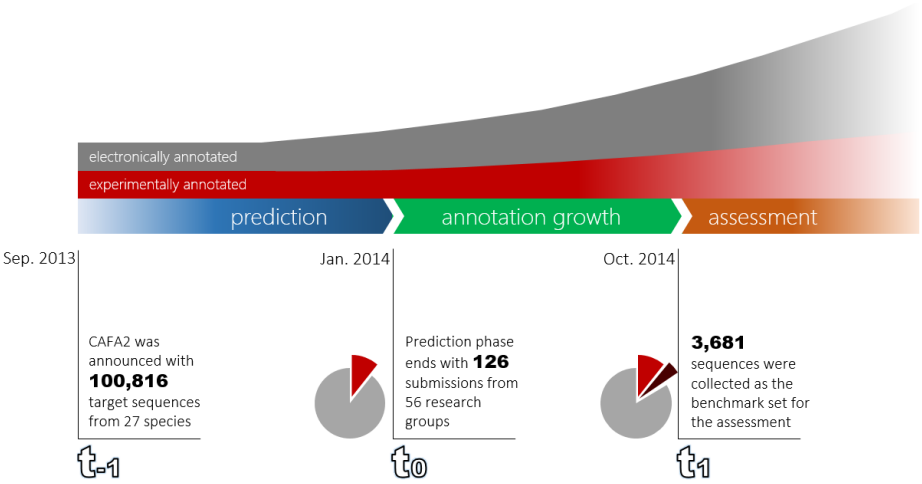


Figure 3: CAFA2 challenge
Extracted from: `http://biofunctionprediction.org/cafa/`

1.3 Deep Learning Basics

Machine Learning is the field of Computer Science that studies automatic methods for making predictions (or, more generally, choosing useful actions) based on past experience of a system. Most of the learning tasks or machine learning fields could be classified in:

- Supervised Learning: uses labeled data. Tasks such as predicting a class or category belong to this field.
- Unsupervised Learning: does not use labeled data. Tasks such as clustering, dimensionality reduction, density estimation or novelty detection belong to this field.
- Semi-supervised Learning: uses partly labeled data. Tasks such as ranking (ordering examples according to some criterion) or reinforcement (delayed rewarding) belong to this field.

This thesis is developed with one goal in mind: use **Deep Learning** to predict protein functions. Deep Learning is a new area of Machine Learning research,

which has been introduced with the objective of moving Machine Learning closer to one of its original goals; Artificial Intelligence, by applying **deep Artificial Neural Networks**. In this project, there will be used two kinds of Neural Networks:

- Artificial Neural Networks (ANN) or Multi-Layer Perceptrons (MLP): the mathematical concepts and the examples seen in this description are extracted from the notes on Neural Networks of Andrej Karpathy[10]. The area of Neural Networks has originally been primarily inspired by the goal of modeling biological neural systems, but has since diverged and become a matter of engineering and achieving good results in Machine Learning tasks. The basic computational unit of the brain is a neuron. Each neuron receives input signals from its dendrites and produces output signals along its (single) axon. The axon eventually branches out and connects via synapses to dendrites of other neurons. In the mathematical model, the signals that travel across the neuron via the axons (for example, x_0) interact multiplicatively with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w_0). The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence (and its direction: excitory (positive weight) or inhibitory (negative weight)) of one neuron on another. In the basic model, the dendrites carry the signal to the cell body where they all get summed. If the final sum is above a certain threshold, the neuron can fire, sending a spike along its axon. In the computational model, we assume that the precise timings of the spikes do not matter, and that only the frequency of the firing communicates information. Based on this rate code interpretation, we model the firing rate of the neuron with an activation function f, which represents the frequency of the spikes along the axon. Historically, a common choice of activation function is the sigmoid function σ, since it takes a real-valued input (the signal strength after the sum) and squashes it to range between 0 and 1.

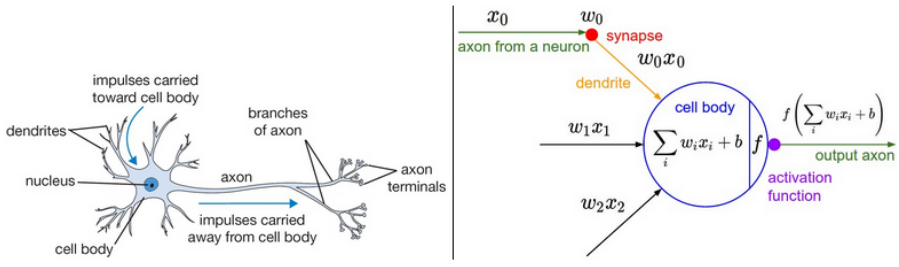


Figure 4: Biological Network (left) and its mathematical model (right) Extracted from: `http://cs231n.github.io/neural-networks-1/`

In consequence, the matrix representation of the mathematical operations that perform this network is: $\sigma(w^T x + b)$ where x represents the vector

of **inputs**, w is the **coefficients** vector, b is the **bias** introduce in each branch and $\sigma = \frac{1}{1+e^{-x}}$ the activation function. With the proper activation function, a single neuron's output can be considered as a linear classifier. This, for instance, could be interpreted as a **Binary Softmax classifier**, interpreting the output of the sigmoid as a probability of one of the classes $P(y_i = 1 \mid x_i; w)$ and the other class $P(y_i = 0 \mid x_i; w) = 1 - P(y_i = 1 \mid x_i; w)$, since they must sum to one. With this interpretation, we can formulate the cross-entropy loss as we have seen in the Linear Classification section, and optimizing it would lead to a Binary Softmax Classifier (also known as **logistic regression**). Since the sigmoid function is restricted to be between $0 - 1$, the predictions of this classifier are based on whether the output of the neuron is greater than 0.5.
Here there are presented two common **non-linearities** or activation functions:

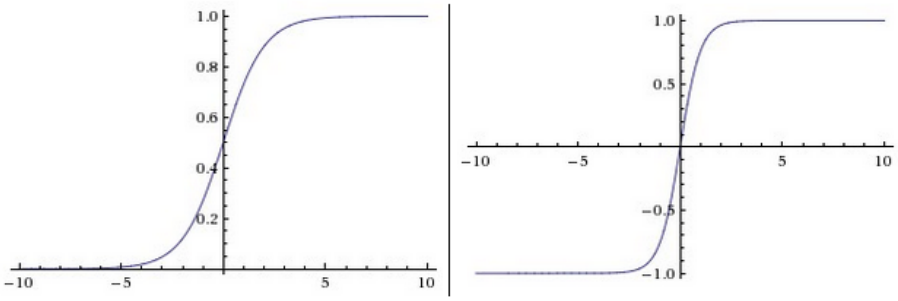


Figure 5: Sigmoid, which squashes real numbers into the interval $[0, 1]$ versus tanh, which squashes real numbers into the interval $[-1, 1]$ Extracted from: `http://cs231n.github.io/neural-networks-1/`

Once understood how a neuron works, it is time to focus on Neural Networks architechtures. Neural Networks are just groups of neurons that are connected in an acyclic graph. These models are usually organized in **layers**. The most common layer type is the **fully-connected layer**, in which neurons between two adjacent layers are fully pairwised connected, but neurons within a single layer share no connections. Here there are presented two examples of fully-connected layers:

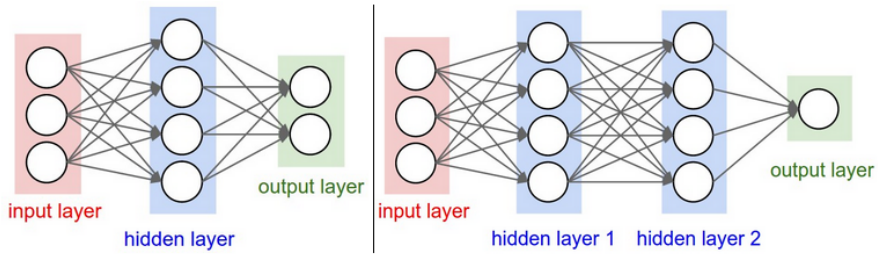


Figure 6: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs (left) versus a 3-layer Neural Network with three inputs, two hidden layers of 4 neurons each and one output layer (right). Extracted from: `http://cs231n.github.io/neural-networks-1/`

The mathematical model of the 2-layer Neural Network is described as $f(W_1x + b)$ and the corresponding model for the 3-layer Neural Network as $f(W_2f(W_1x + b_1) + b_2)$ where x is the vector of inputs, f is the non-linearity or the activation function, W_1 are the coefficients of the first layer, b_1 is the bias introduced by the first layer and respectively W_2 and b_2 are the coefficients and the bias of the second layer.
In order to measure the **complexity** of a Neural Network, usually there are taken into account the number of units or even more commonly the number of parameters. Analyzing the example presented in figure 6:

- The 2-layer Neural Network has $4+2 = 6$ neurons, $[3x4]+[4x2] = 20$ weights and $4+2 = 6$ biases, in consequence the total is 26 learnable parameters.
- The 3-layer Neural Network has $4+4+1 = 9$ neurons, $[3x4]+[4x4]+[4x1] = 32$ weights and $4+4+1 = 9$ biases, for a total result of 41 learnable parameters.

As a result, it can be assumed that, in this example, the 3-layer Neural Network is more complex than the 2-layer. As the number of neurons and the number of layers used to face a problem are increased, the capacity of the network is also increased. That means that the network could express more complicated functions. Increasing the capacity will easily lead to **overfitting** (it is too complex to fit the data) while not providing the network with enough capacity will cause **underfitting** (it is not complex enough to fit the training data).

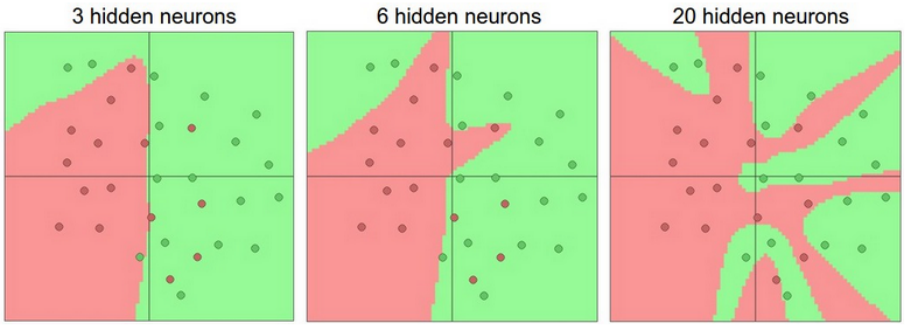


Figure 7: Example of how increasing the number of neurons leads to overfitting. Extracted from: `http://cs231n.github.io/neural-networks-1/`

However, instead of controlling the number of layers and the number of units used to prevent overfitting, other techniques are prefered such as L2 regularization, dropout or input noise, which reduce complexity of the network by imposing conditions in the learnable parameters. In any case, it should be studied which architechture gives a better **generalization** of the data in the **validation and test set**.

- Convolutional Neural Networks (CNN): the mathematical concepts and the examples seen in this description are extracted from the notes on Convolutional Neural Networks of Andrej Karpathy[11]. These kind of architectures make the explicit assumption that the inputs are structured data or images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.
  ANN described before do not scale well to full images. In CIFAR-10 (the CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.) [21], using images that have as size only 32x32x3 (32 wide, 32 high and 3 color channels), a single fully-connected unit in a first hidden layer of a MLP would have $32x32x3 = 3072$ weights. If an image of more respectable size was used as input, such as 200x200x3, this would lead to $200x200x3 = 120000$ weights. Clearly, this full connectivity introduces too much complexity for this kind of data and quickly leads to overfitting. CNN have neurons arranged in 3 dimensions: width, height, depth. Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network. In addition, CNN architecture will reduce the full image into a single vector of class scores, arranged along the depth dimension.

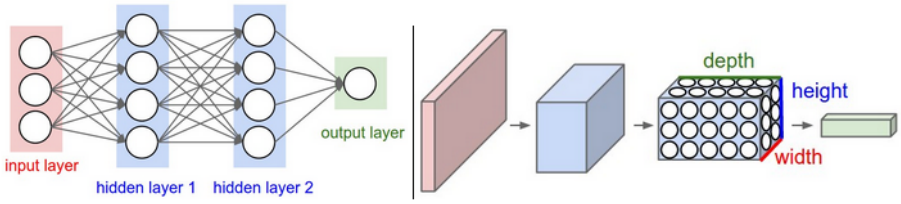


Figure 8: Ordinary Neural Network (left) versus Convolutional Neural Network. A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels). Extracted from: `http://cs231n.github.io/convolutional-networks/`

Let's focus on the example of using the images of CIFAR-10 as input. Here it is presented an overview of an architecture used for CIFAR-10 classification. The sequence of layers used are explained here in detail:

- Input: raw pixel values of the image of dimensions [32x32x2] (width 32, height 32 and 3 color channels: R, G, B.)
- CNN layer: computes the output of neurons that are connected to local regions of the input, each computing a dot product between their weights and a small region they are connected to in the input volume. The resultant volume will depend on the number of filters used for this layer and their size.
- Relu: activation function applied elementwise, such as $max(0, x)$, leaving the volume unchanged.
- Pool: performs a downsampling operation along the spatial dimensions (width, height), reducing the volume size.
- Fully-Connected: layer that will compute the class scores, resulting in volume of size [1x1x10] due to the fact that the 10 numbers correspond to a class score, according to the 10 categories of CIFAR-10. Each unit in this layer will be connected to all the neurons in the previous volume.

With this procedure, a CNN transforms the original image layer by layer from the original pixel values to the final class scores. The CNN layer implements **local connectivity**. Dealing with high-dimensional inputs such as images, it is wasteful and impractical to connect neurons to all units of the previous volume. Instead, they are connected to a local region of the input volume. It is important to emphasize that the connections are **local in space** (in this example, width and heigth) but always **full along the entire depth** of the input volume (the neurons always processed the

3 channels, R, G, B).
In the CNN layer, there are three hyperparameters that control the size of the output volume: **number of filters**, **stride** and size of **zero-padding**:

- The number of filters corresponds to the depth of the output volume of the layer. Each filter will look for something different in the input. It is very common in image analysis that filters focus on oriented edges or blobs of color.
- The stride which the filter is slided should be specified. If the stride is 3, the filter jump 3 pixels at a time as they are slide along the image. A bigger stride will produce smaller output volumes spatially.
- Zero-padding consists of conveniety padding the input volume with zeros. This allows to control the spatial size of the output volumes.

To calculate the spatial size of the output volume of a CNN layer as a function to the input volume this mathematical formula is used: $(W - F + 2P)/S + 1$ where W is the volume size, F the filter size, P the amount of zeros added on the border and S the stride.
Parameter sharing scheme allows CNN to control and reduce the number of parameters, just by making one reasonable assumption: if one feature is useful to compute at some spatial point (x,y), then it should also be useful to compute at a different position (x2, y2). Therefore, a filter used in a depth slice will has the same coefficients for this depth slice, regardless the spatial position.

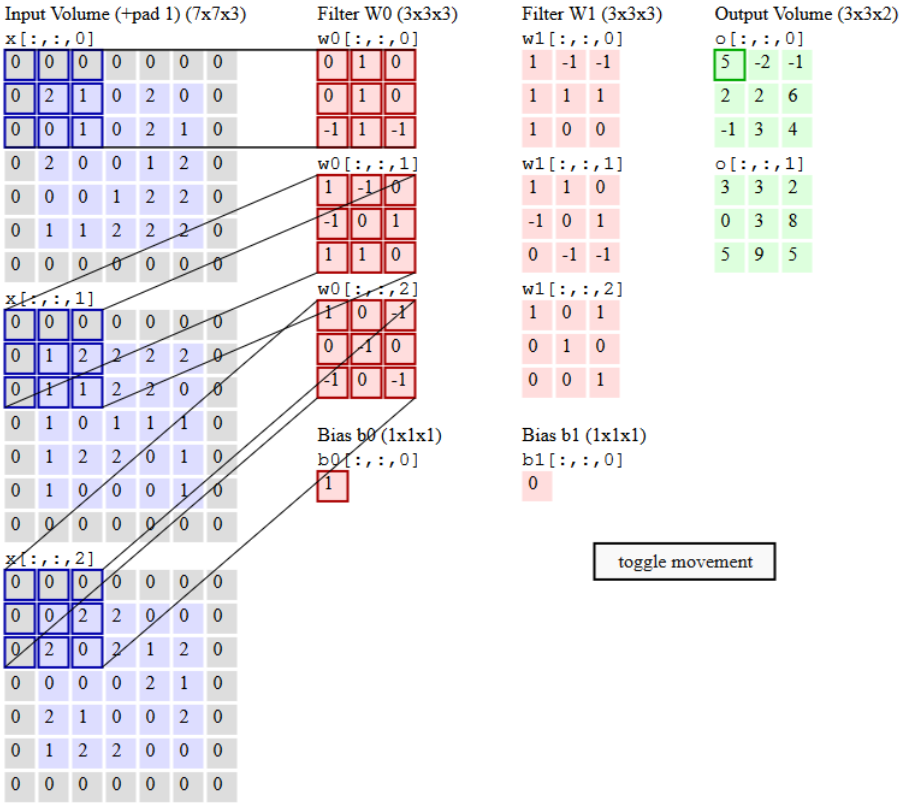


Figure 9: Example of how a Convolutional Layer works. The input volume has size 7x7x3, 2 filters of size 3x3 are used, with stride equal to 1 and zero-padding equal to 1 is used. This architecture results into an output volume of 3x3x2.
Extracted from: `http://cs231n.github.io/convolutional-networks/`

> In figure 9, a toy example is presented. Each neuron has $3x3x3 = 27$ connections (connected to a local spatial region of 3x3 but to all depths, 3). In each slice, the coefficients of the filter are shared for all the neurons, therefore for all the architecture there are just $3x3x3x2 = 54$ weights calculated and $2x1 = 2$ bias terms. In consequence, just 18 neurons are needed to produce this model. In comparison with a fully-connected layer, just a neuron will have $7x7x3 = 147$ weights and 1 bias term.

Among the methods presented in CAFA2, Deep Learning was rarely use. Therefore this project is a great chance to see how a method based on a Deep Learning architecture performs in this area, and, with the proper data as input and the proper tuning, whether it could outperform the other methods presented in CAFA2.

1.4 From the point of view of Data Science

In terms of Data Science, the prediction of protein functions is considered **Supervised Learning** because the data is labeled. It is called supervised learning because the process of an algorithm learning from the training dataset can be thought of as a teacher supervising the learning process. The correct answers or the truth is known, the algorithm iteratively makes predictions on the training data and is corrected by the teacher. Learning stops when the algorithm achieves an acceptable level of performance.
As the output variable is a set of categories for each individual, this is considered a **classification** problem, and specifically **Multi-label classification**. This can be thought as predicting properties of a data-point that are not mutually exclusive, such as topics that are relevant for a document. A text might be about any of religion, politics, finance or education at the same time or none of these.
In addition, this prediction is considered a **structured-output learning** task that involves predicting structured objects, such as images, text and molecules or chemical structures, where the data consists of several parts, and not only the parts themselves contain information, but also the way in which the parts belong together.

1.5 Libraries used: Theano and Lasagne

Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently.
Lasagne is a lightweight library to build and train neural networks in Theano. Building a deep Neural Network using Lasagne is as simple as this:

Neural Networks in Lasagne

```
inputs  = theano.tensor.tensor4()  # 4-dimensional symbolic array
targets = theano.tensor.tensor4()  # 4-dimensional symbolic array
nchannels = 3
network = lasagne.layers.InputLayer(shape=(None, nchannels, None, None), input_var=inputs)
network = lasagne.layers.Conv2DLayer(network, num_filters=128, filter_size=(3,3))
network = lasagne.layers.Pool2DLayer(network, pool_size=(2,2))
network = lasagne.layers.Conv2DLayer(network, num_filters=256, filter_size=(3,3))
network = lasagne.layers.GlobalPoolLayer(network, pool_function=theano.tensor.max)
network = lasagne.layers.DenseLayer(network, num_units=1024)
network = lasagne.layers.DropoutLayer(network, p=0.5)
network = lasagne.layers.DenseLayer(network, num_units=1)
# optional arguments: nonlinearity, weight initialization, stride, pad, ...

predictions      = lasagne.layers.get_output(network)                          # symbolic
trainable_params = lasagne.layers.get_all_params(network, trainable=True)      # symbolic
loss             = lasagne.objectives.squared_error(predictions, targets).mean()  # symbolic
adam_updates     = lasagne.updates.adam(loss, trainable_params, learning_rate=0.01)
train_fn = theano.function([inputs, targets], [predictions, loss], updates=adam_updates)

for i in range(10000):                                  # mini-batch training loop
    X,Y = my_minibatch_producer.get()                   # load data
    minibatch_predictions,minibatch_loss = train_fn(X,Y)  # train
```

Figure 10: Example of how to build a Neural Network using Lasagne and Theano

Lasagne allows you to define an arbitrarily structured neural network by creating and stacking or merging layers. Since every layer knows its immediate incoming layers, the output layer (or output layers) of a network double as a handle to the network as a whole, so usually this is the only thing we will pass on to the rest of the code. In the example presented in figure 10, the resultant model is built:

Table 1: Model developed with Lasagne and Theano in figure 10. The inputs variable is a Theano tensor of 4 dimensions (batchsize, channels, image height and image width), matching the input size of the network and the targets variable is also a Theano tensor of 4 dimensions. Notice that in case dropout is applied, it should be specified after the layer where it is going to be applied.

Input Layer (batchsize, channels, image height, image width)
Convolutional 2D Layer: filter size = 3x3, number of filters = 128
Pool 2D Layer: pool size 2x2
Global Pool Layer
Dense Layer with dropout: number of neurons = 1024
Dense Layer: number of neurons = 1

2 Data

The data used for the project is **Swissprot** the manually annotated and reviewed section of the UniProt Knowledgebase (UniProtKB). It is a high quality annotated and non-redundant **protein sequence database**, which brings together experimental results, computed features and scientific conclusions. Since 2002, it is maintained by the UniProt consortium and is accessible via the UniProt website.
I have analyzed in this data **62354 proteins**. Two main fields of analysis are considered: **sequences** and **GOs**.

2.1 Sequences

Regarding the sequences, they are represented with **25 characters**: A, B, C, D, E, F, G H, I, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z. The **average lenght** of a sequence is approximately **530 characters** and most of them are of length less than 2000 characters (about 98.33%). Here there are presented histograms: all the sequences, sequences with a length smaller than 2000 characters, sequences with the longest lenghts and sequences with the shortest lengths.

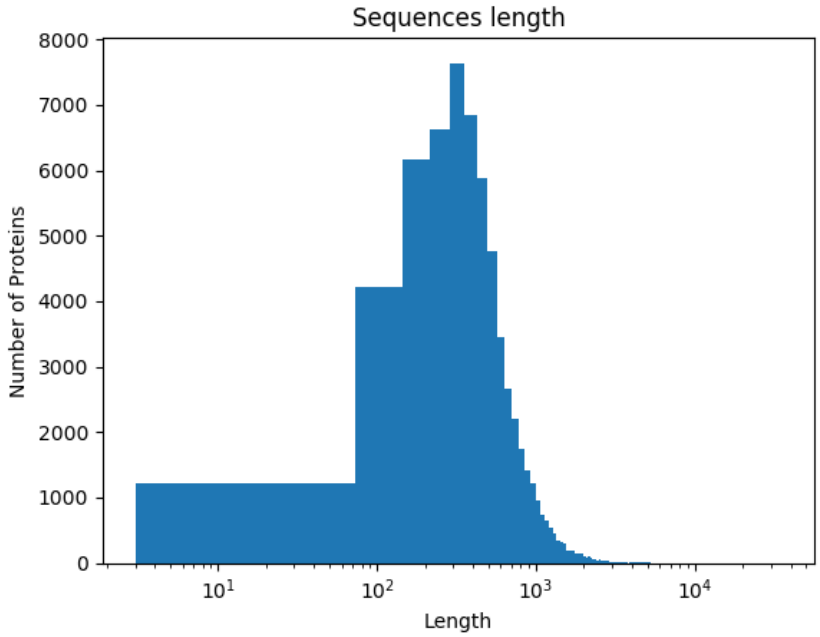


Figure 11: Distribution of the sequences length

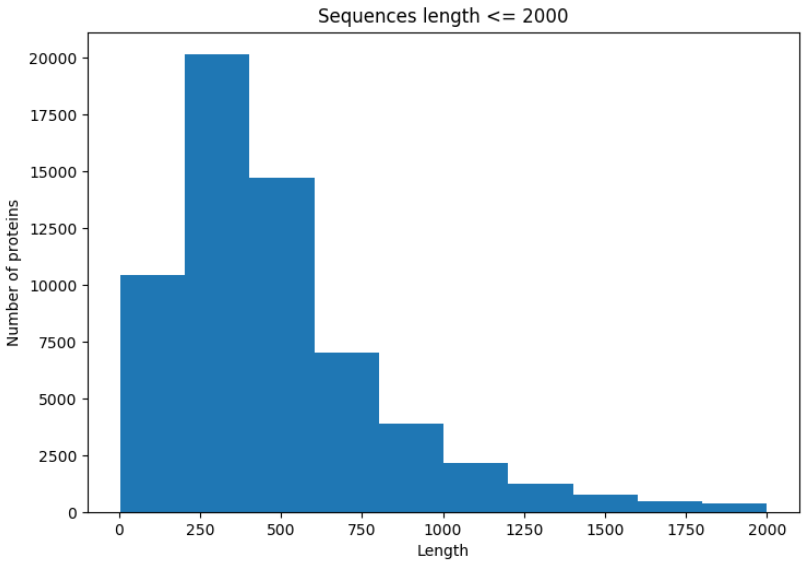


Figure 12: Sequences with length less than 2000

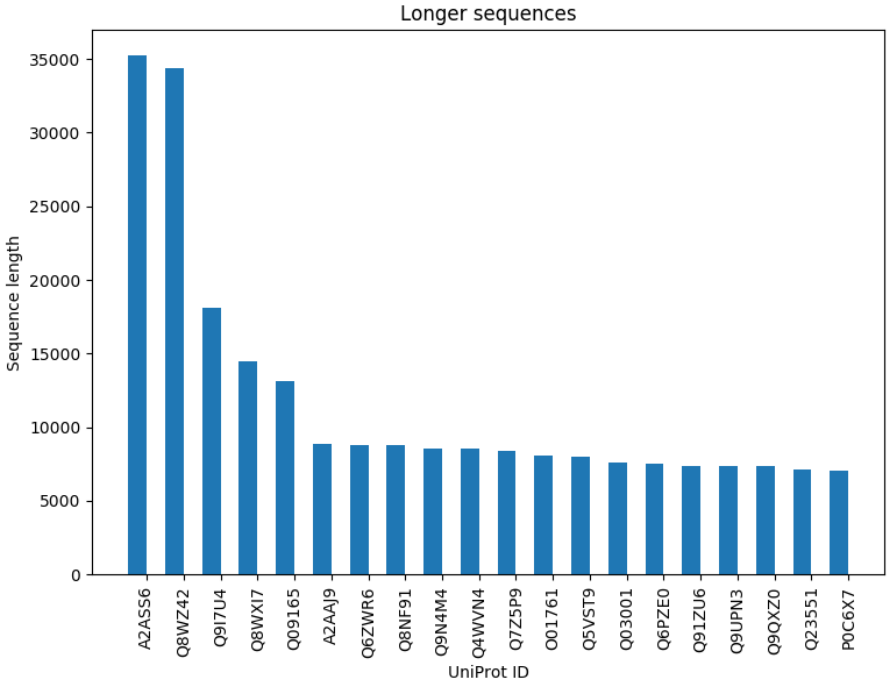


Figure 13: Longest sequences

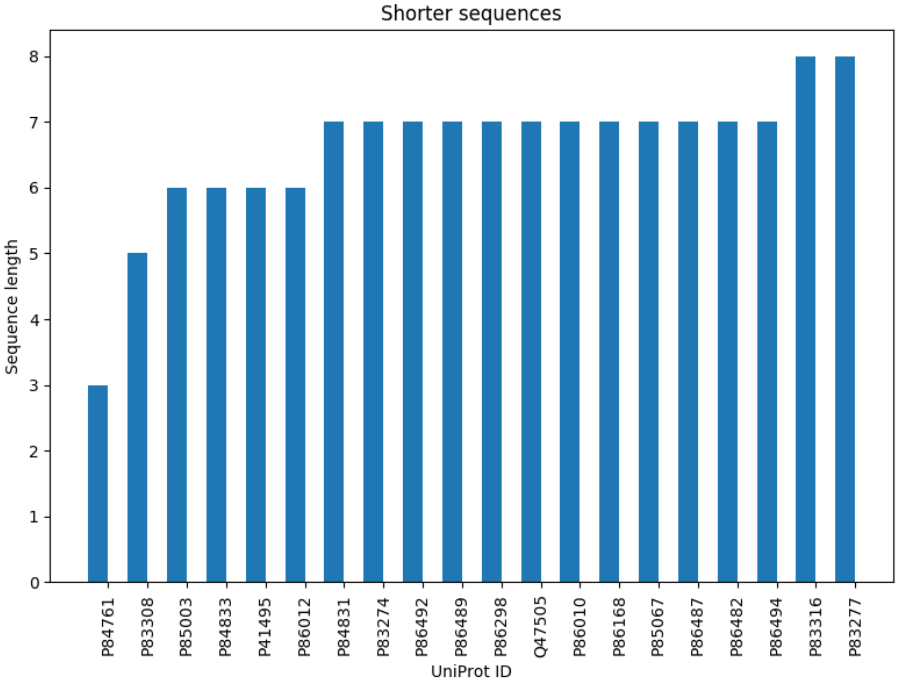


Figure 14: Shortest sequences

These proteins that have so short sequences are called **oligopeptides**.

2.2 Gene Ontology (GO)

Regarding the GOs, I have divided the analysis by categories:

- BPO: There are 16412 proteins without BPO. However, there are 188957 BPOs in Swissprot from which just 14175 are unique. This gives an average BPO per protein of 3.03. Here I present a histogram of the probabilities of BPOs, showing just the most common BPOs:

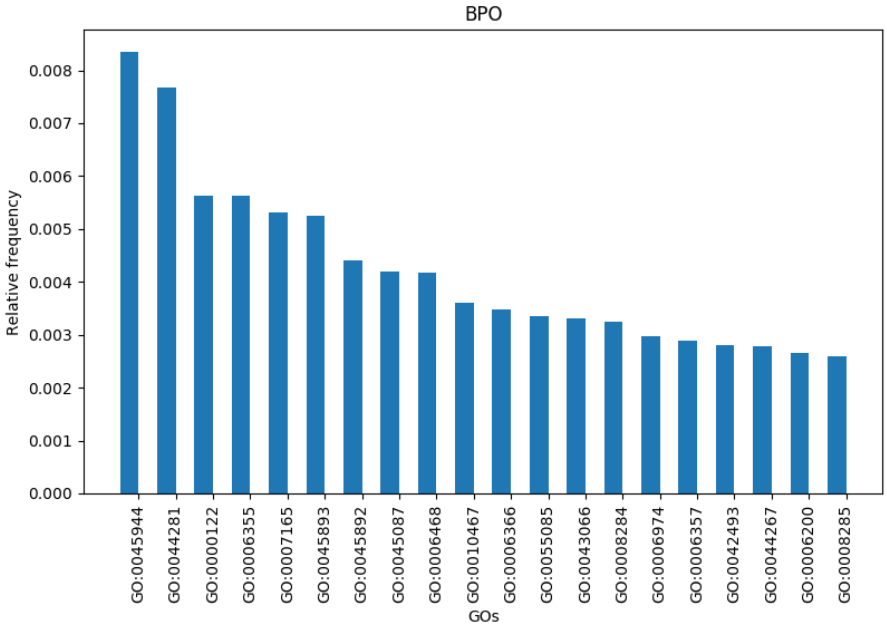


Figure 15: Biological Process Ontology (BPO) probabilities

- CCO: There are 18491 proteins without CCO. However, there are 96740 CCOs in Swissprot from which just 1881 are unique. This gives an average CCO per protein of 1.55. Here I present a histogram of the probabilities of CCOs, showing just the most common CCOs:

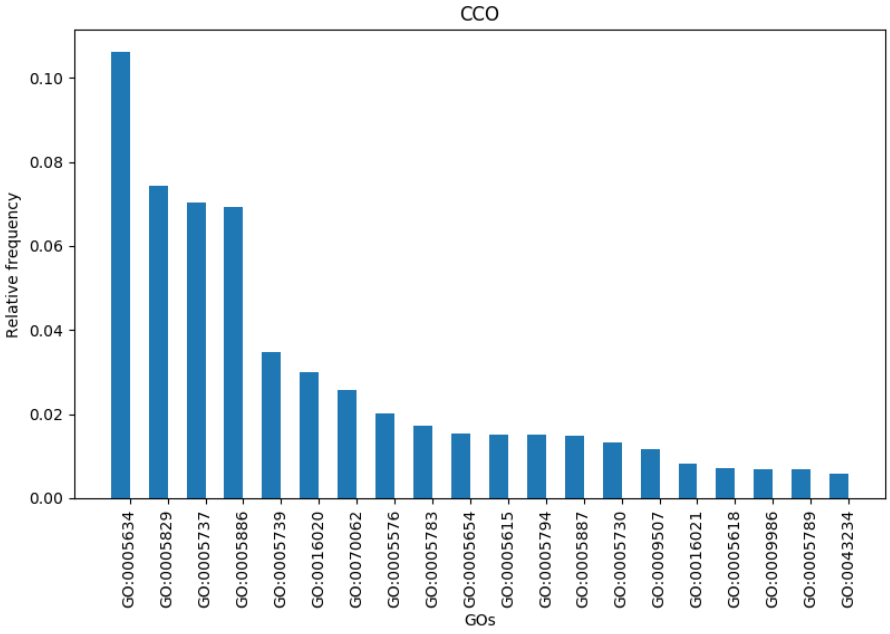


Figure 16: Cellular Component Ontology (CCO) probabilities

- MFO: There are 23688 proteins without MFO. However, there are 75037 MFOs in Swissprot from which just 5435 are unique. This gives an average MFO per protein of 1.2. Here I present a histogram of the probabilities of MFOs, showing just the most common MFOs:

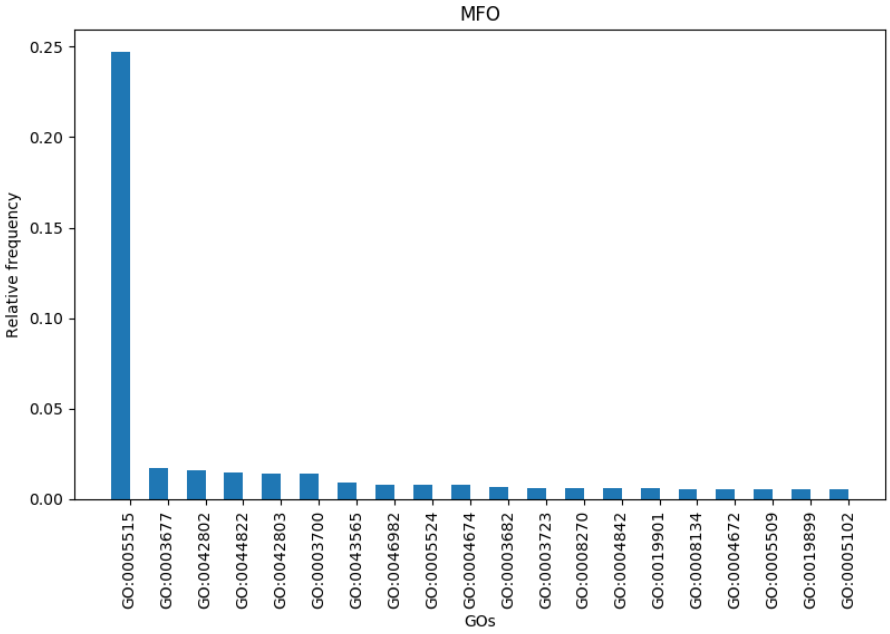


Figure 17: Molecular Function Ontology (MFO) probabilities

3 Models

3.1 Naive

This is one of the simplest models that could be used. The only information used is the probabilities of the GOs. For each GO category (BPO, CCO and MFO) the **probabilities of each GO** term are calculated and then they are assigned to the test proteins by chance, using a **Bernuolli Distribution** for each GO term:

$$f(k;p) = p^k(1-p)^{1-k}$$

where p is the probability of 1 GO term in its respective category and k is the number of trials. Therefore this model needs almost no **training** at all, just to calculate the probabilities that each GO term appears as a protein function in a training sample. The idea is to set a fixed number of trials k and, for each GO term, count the number of successes; if this is bigger than $k/2$, the GO term is assigned to the test protein. Yoy may notice that no information about the proteins is used, just statistical information about the GO terms.

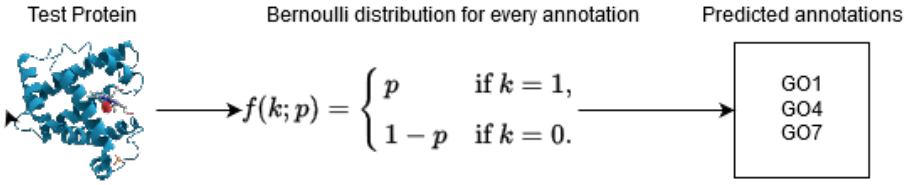


Figure 18: How the annotations of a test protein are predicted using the Naive model. Function f maps for each GO term the probability of appearance (case $k = 1$, with probability p) or the probability of no appearance ($k = 0$, with probability $1 - p$)

3.2 Basic Local Alignment Search Tool (BLAST)

By using BLAST, regions of **similarity between biological sequences** are found. The program compares nucleotide or protein sequences to sequence databases and calculates the statistical significance.

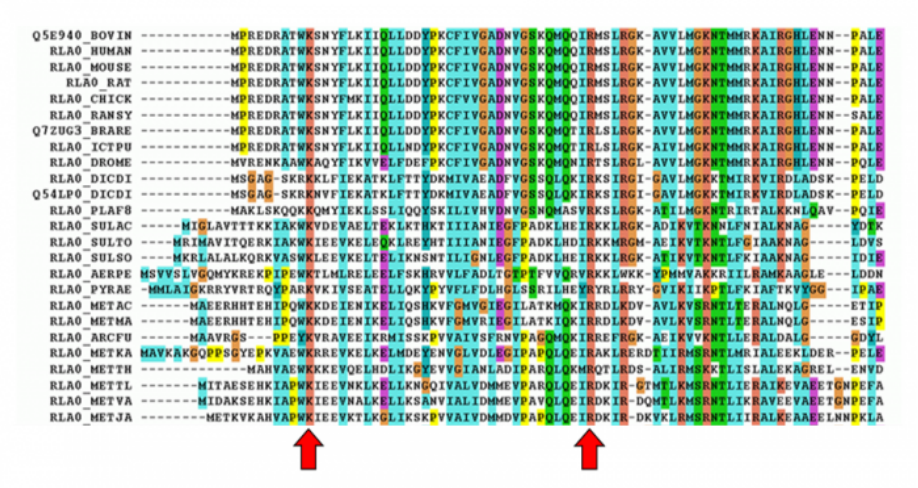


Figure 19: Example of a BLAST run. Extracted from:
`https://www.ebi.ac.uk/training/online/course/introduction-protein-classification-ebi/what-are-protein-signatures/how-do-protein-signatures`

Regarding the **training**, this model needs no training, it can be executed on the run. Given a test protein, it just searchs for the **most similar protein** and assume that they have the **same functions**. Therefore the functions of the most similar protein (in all GO categories) are assigned to the test protein.

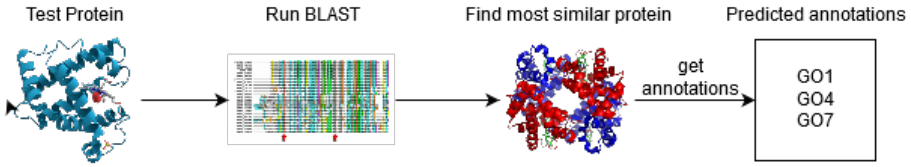


Figure 20: How the annotations of a test protein are predicted using the BLAST model

3.3 Convolutional Neural Network (CNN)

As protein sequence is considered **structured data**, therefore is a priori a good idea to try to deal with this problem using CNNs, which focus on analyzing only a **local region** of the input data at a time. Here it is described the process of preparing the data, training the model and using it to predict the functions of the test proteins:

1. Preprocessing: For this model, the **Position-Specific Scoring Matrix** (PSSM) will be used as input. A PSSM is a matrix of dimensions $A \times L$

which has as **rows** the **elements of the alphabet** (A) and as **columns** the **positions** (L) **in the pattern**. The values of the PSSM are the **probabilities** of having an specific element of the alphabet in a particular position of the sequences. Here it is presented an example of PSSM for a given set of DNA sequences (which is simpler to represent than amino acids PSSM because they only use 4 characters):

GAGGTAAAC
TCCGTAAGT
CAGGTTGGA
ACAGTCAGT
TAGGTCATT
TAGGTACTG
ATGGTAACT
CAGGTATAC
TGTGTGAGT
AAGGTAAGT

Figure 21: Example of DNA sequences

$$M = \begin{array}{c} \\ A \\ C \\ G \\ T \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \left[\begin{array}{ccccccccc} 0.3 & 0.6 & 0.1 & 0.0 & 0.0 & 0.6 & 0.7 & 0.2 & 0.1 \\ 0.2 & 0.2 & 0.1 & 0.0 & 0.0 & 0.2 & 0.1 & 0.1 & 0.2 \\ 0.1 & 0.1 & 0.7 & 1.0 & 0.0 & 0.1 & 0.1 & 0.5 & 0.1 \\ 0.4 & 0.1 & 0.1 & 0.0 & 1.0 & 0.1 & 0.1 & 0.2 & 0.6 \end{array}\right] \end{array}$$

Figure 22: Example of a Position-Specific Score Matrix (PSSM)

With the proteins data, precomputed **Multiple Sequence Alignments** (MSA) will be used, which represent protein clusters. These files are available in the Computer Vision department data. Here it is shown an example of an MSA file:

```
>Q2V2P9 GO:0005743
MFFSQVLRSSARAAPIKRYTGGRIGESWVITEGRRLIPEIFQWSAVLSVCLGWPGAVYFFSKARKA
>tr|G8ZX67|G8ZX67_TORDC Uncharacterized protein OS=Torulaspora delbrueckii (strain ATCC 10662 / CBS 1146 / NBRC 0425
MFYTQVLRSAAAAARA--TSRNHNSQSWAVSEAKRLTPAILGWGGFLAGCLGWPFAIKHFMSAN--
>tr|A0A0L8RP72|A0A0L8RP72_SACEU COX26-like protein OS=Saccharomyces eubayanus GN=DI49_0313 PE=4 SV=1
MFFNQVLRSSARAAPIKRYTGGRVGEAWMITESRRLVPEVFQWGAAICTCLGWPGAVYFFSKAR--
>tr|A0A0D4Q0F6|A0A0D4Q0F6_YEASX Cox26p OS=Saccharomyces cerevisiae YJM1402 GN=COX26 PE=4 SV=1
MFFSQVLRSSARAAPIKRYTGGRIGESWVITEGRRLIPEIFQWSAVLSVCLGWPGAVYFFSKAR--
>tr|A0A0D4N531|A0A0D4N531_YEASX Cox26p OS=Saccharomyces cerevisiae YJM1129 GN=COX26 PE=4 SV=1
MFFSQVLRSSARAAPIKRYTGGRIGESWVITEGRRLIPEIFQWSAVLSVCLGWPGAVYFFSKAR--
>tr|A0A0D4QCZ8|A0A0D4QCZ8_YEASX Cox26p OS=Saccharomyces cerevisiae YJM1434 GN=COX26 PE=4 SV=1
MFFSQVLRSSARAAPIKRYTGGRIGESWVITEGRRLIPEIFQWSAVLSVCLGWPGAVYFFSKAR--
>tr|A0A0D4P9B3|A0A0D4P9B3_YEASX Cox26p OS=Saccharomyces cerevisiae YJM1342 GN=COX26 PE=4 SV=1
MFFSQVLRSSARAAPIKRYTGGRIGESWVITEGRRLIPEIFQWSAVLSVCLGWPGAVYFFSKAR--
>tr|A0A0D4QXG6|A0A0D4QXG6_YEASX Cox26p OS=Saccharomyces cerevisiae YJM1479 GN=COX26 PE=4 SV=1
MFFSQVLRSSARAAPIKRYTGGRIGESWVITEGRRLIPEIFQWSAVLSVCLGWPGAVYFFSKAR--
>tr|A0A0D4N2E2|A0A0D4N2E2_YEASX Cox26p OS=Saccharomyces cerevisiae YJM1083 GN=COX26 PE=4 SV=1
MFFSQVLRSSARAAPIKRYTGGRIGESWVITEGRRLIPEIFQWSAVLSVCLGWPGAVYFFSKAR--
>tr|A0A0D4N154|A0A0D4N154_YEASX Cox26p OS=Saccharomyces cerevisiae YJM996 GN=COX26 PE=4 SV=1
MFFSQVLRSSARAAPIKRYTGGRIGESWVITEGRRLIPEIFQWSAVLSVCLGWPGAVYFFSKAR--
>tr|A0A0D4MIB6|A0A0D4MIB6_YEASX Cox26p OS=Saccharomyces cerevisiae YJM693 GN=COX26 PE=4 SV=1
MFFSQVLRSSARAAPIKRYTGGRIGESWVITEGRRLIPEIFQWSAVLSVCLGWPGAVYFFSKAR--
>tr|A0A023ZAL5|A0A023ZAL5_YEASX Cox26p OS=Saccharomyces cerevisiae YJM993 GN=COX26 PE=4 SV=1
MFFSQVLRSSARAAPIKRYTGGRIGESWVITEGRRLIPEIFQWSAVLSVCLGWPGAVYFFSKAR--
>tr|C7GIM7|C7GIM7_YEAS2 YDR119W-A-like protein OS=Saccharomyces cerevisiae (strain JAY291) GN=C1Q_00043 PE=4 SV=1
MFFSQVLRSSARAAPIKRYTGGRIGESWVITEGRRLIPEIFQWSAVLSVCLGWPGAVYFFSKAR--
>tr|A0A0D4QIG2|A0A0D4QIG2_YEASX Cox26p OS=Saccharomyces cerevisiae YJM1444 GN=COX26 PE=4 SV=1
MFFSQVLRSSARAAPIKRYTGGRIGESWVITEGRRLIPEIFQWSAVLSVCLGWPGAVYFFSKAR--
```

Figure 23: Example of Multiple Sequence Alignment (MSA) file of UniProt ID Q2V2P9

In this example, the representant of this cluster is the Uncharacterized protein YDR119W-A, with UniProt ID Q2V2P9. The first line of each MSA presents always the representant protein of the cluster plus the GO terms associated with this cluster. Notice that for the purpose of this project, only clusters with at least 90% of sequence similiraty, available in **UniProt Reference Clusters** (UniRef), and all the proteins listed share at least all the annotations related to this MSA. Therefore it is assumed that the proteins that belong to the **same cluster share functions**. Using these clusters the PSSMs will be generated. In order to calculate a PSSM, a complex script was developed due to the fact that the format of the MSA files (.a3m) provided was not compatible with the standard ways to compute PSSMs ready for using in libraries such as Biopython. The faster idea that was found consists of **reading the file line by line and process this line on the go**:

- First line is read and the corresponding GO terms related to the cluster are extracted.
- Other lines are read and for each one, the sequence is processed to start building the PSSM, taking into account the positions in which the characters of the sequence appear. In addition, the UniProt IDs of the proteins are stored to easily associated them with the cluster.

This process is very **time-consuming** because of the big size of some MSA files. The biggest file found is about 370MB and contains more than 131000 lines. You may notice that not all the PSSMs will have the same dimensions because of the different lenght of the sequences that appear in the MSAs.
Respecting the targets, the GOs will be encoded using **k-hot encoding**.

This means that they are represented as a vector of length k, the number of different GOs, and with 1 and 0 as possible values for each position, meaning 1 that a protein has this GO as a function or 0 otherwise. Here I present a toy example:

$$Prot1 = \begin{bmatrix} \overset{GO_1}{1} & \overset{GO_2}{0} & \overset{GO_3}{0} & \overset{GO_4}{1} & \overset{GO_5}{0} & \overset{GO_6}{0} & \overset{GO_7}{1} \end{bmatrix}$$

where k=7 (there are 7 possible annotations for a protein) and $Prot1$ has GO_1, GO_4 and GO_7 as its functions.

2. Training: due to the deep learning nature of the model, the process of training is far more complicated than the other models. First of all, in order to get an idea of which should be the optimal values of the hyperparameters, an small subset of the whole data available is exploited: the first trained models work with just **4 PSSMs** and must predict annotations among **10 GO terms**. Then the same model was tested with **17 PSSMs** and **67 GO terms** different GO terms and finally with **all the data available** (10000 PSSMs), with more than **12000 GO terms**. In this case, the model is trained through **batches and epochs**. An epoch is the whole training set and a batch is a subset of the training data that is taken together in the process of training. The main idea is train the model through different number of epochs until its behaviour does not improve. To detect this point, after each epoch several **evaluation metrics** (section 4) are checked, in both training set and validation set. Therefore, to avoid unnecessary steps, a particular version of **Early Stopping** is implemented: if validation loss does not decrease in 10% of the total number of epochs consecutively, the training process is interrupted and the best parameters found are returned. The principal goal is to achieve a low **training and validation loss**, although other measures are used in addition to get a full view of how well the model is performing. The definition of an appropiate training loss is crucial in the training process, because it will be what will guide the model in the learning process. For this problem, **binary crossentropy** is chosen for the task. For a given protein, here it is shown how to calculate it:

$$L = \frac{\sum_{i=1}^{k} -y_i \log y_i^{pred} - (1 - y_i) \log (1 - y_i^{pred})}{k}$$

Althoug binary crossentropy is really useful for this problem, it does not really give us an specific association of test protein and its corresponding function; it only outputs probabilities for exery protein annotation. For that result a way to convert this probabilities cinto 1's (the test protein has the GO term) or 0's (the test protein has not the GO term) should be developed. Thus, after analyzing the training data, the **average number of protein functions** per cluster is calculated and this number is used to set the largest values output by binary crossentropy to 1 and the rest to 0. Here it is presented an example of the training and validation process:

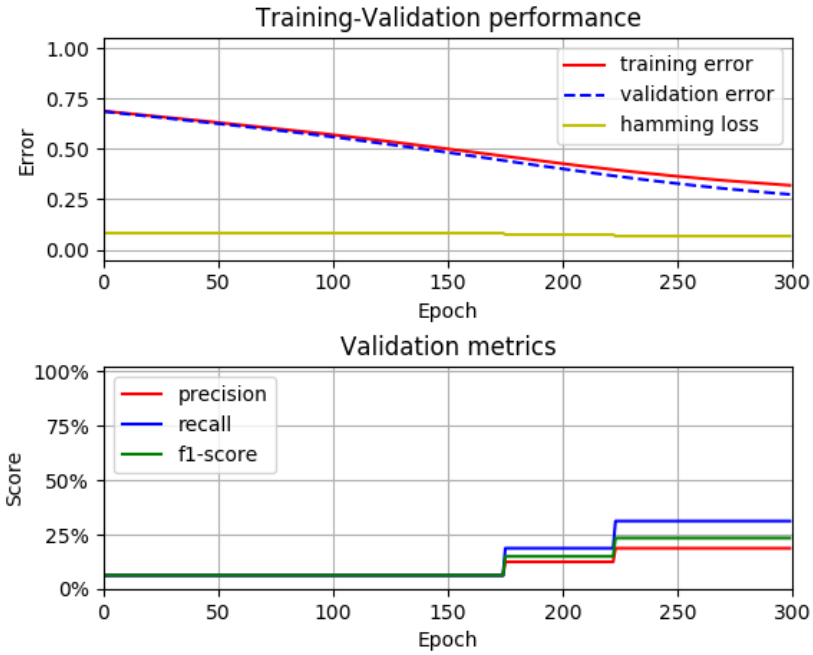


Figure 24: Example of training and validation of the model

Table 2: Deep Learning Model based on Convolutional Neural Networks with 1 Convolutional Layer. The input dimensions are (batchsize, A, L) where A is the alphabet size and L is the sequence lenght that appears in the MSA file. We are forced to use a batchsize of 1 due to the different dimensions of the PSSMs, that depend on the sequence length of each MSA, L, to avoid dimensionality problems. The number of units in the Dense Layer is equal to the number of GO terms taken into account in the analysis. For that reason, using the subset of 4 PSSMs as input, there are considered 10 GO terms so in consequence there are used 10 neurons for the Dense Layer.

Input Layer (1,26, L)
Convolutional Layer: filter size = 3, number of filters = 100
Global Pool Layer
Dense Layer: number of units = 10

This model is characterized by this set of parameters:

- Data used: 17 PSSMs and 67 different GO terms
- Random data split: 80% training and 20% validation
- Number of CNN layers: 1

- Filter size: 3
- Number of filters used: 100
- Learning rate used: 0.001
- Number of epochs used: 300
- Batchsize used: 1

You may notice that although the training and validation loss decrease as more epochs are input, hamming loss remains more or less constant and the precision, recall and F1 score never achieve a great score (they always remain below 32%). This happens because of the nature of the problem, where there is an important **unbalance in the data labels**. An average protein cluster has only **4 GO terms** out of the **67** possible GO terms (in the case of working with just 17 PSSMs). Therefore the k-hot encoded target vector is really **sparsed**, having usually four 1's and the rest 0's. This means that the easiest thing to do for the model is just to predict that a protein cluster has no function at all (prediction vector of all 0's), making approximately less than a 6% error in each prediction by predicting that all the test proteins have no function at all, although this is a trivial and useless answer. To solve this situation, the loss function should be modified in order to give more importance to predict correctly the 1's (the real function prediction) and less to predict 0's (the functions that are not associated test proteins).

3. Prediction: to predict the function of a test protein, it is necessary to previously find to which cluster belongs. Once this is done, the corresponding precomputed PSSMs is used as input for the model and the output, the protein functions predicted for this cluster, are the ones associated to the test protein.

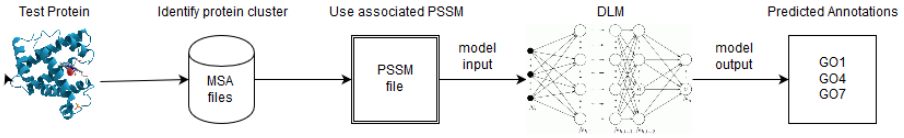


Figure 25: How the annotations of a test protein are predicted using the CNN model

4 Evaluation

- Hamming loss (HL): is calculated as the fraction of labels that are incorrectly predicted:

$$HL = \frac{\sum_{i=1}^{k} y_i^{pred} \oplus y_i}{k}$$

where y is the vector of size k (number of different GOs) that represents the k-hot encoding for each protein function studied with the true functions of a given protein and y^{pred} is the corresponding prediction of these functions in the same format.

- Precision (P): is calculated as the number of true positives (T_{p}) over the number of T_{p} plus the number of false positives (F_{p}):

$$P = \frac{T_p}{T_p + F_p}$$

- Recall (R): is calculated as the number of $\mathrm{T_p}$ over the number of $\mathrm{T_p}$ plus the number of false negatives ($\mathrm{F_n}$):

$$R = \frac{T_p}{T_p + F_n}$$

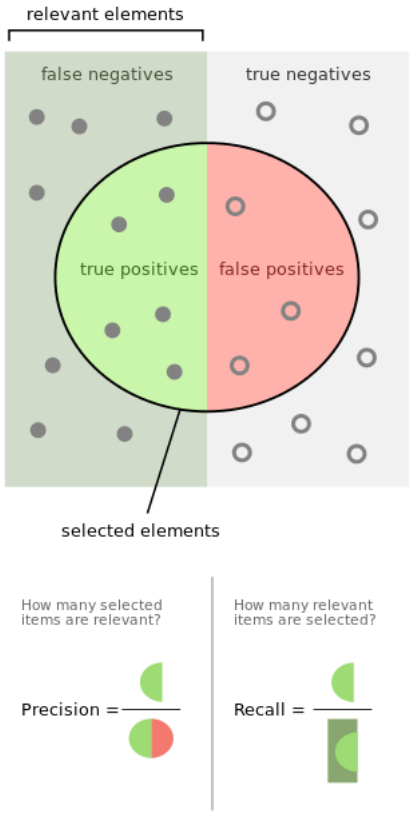


Figure 26: Summary of how to calculate Precision and Recall. Extracted from: `https://en.wikipedia.org/wiki/Precision_and_recall`

- F1 score (F1): is calculated as a weighted average of the P and R. It reaches its best value at 1 and worst score at 0.

$$F1 = 2 \cdot \frac{P \cdot R}{P + R}$$

5 Results

First of all, CAFA2 results are presented below, where the performance of Naive and BLAST could be checked.

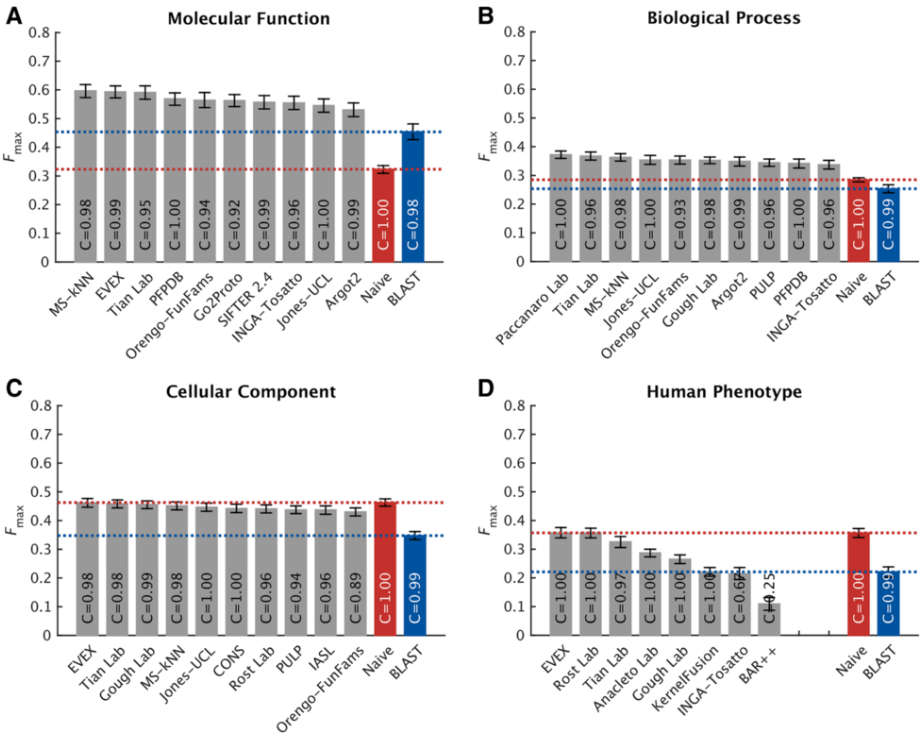


Figure 27: JIANG, Yuxiang, et al. An expanded evaluation of protein function prediction methods shows an improvement in accuracy. Genome biology, 2016, vol. 17, no 1, p. 184. CAFA2 results. In this challenge, as an evaluation metric, F1 score is used in each GO term cathegory. Yoy may notice that the different model submitted behave really different depending on which annotation type they are trying to predict.

You may notice that in CAFA2, all the methods used were **focused on different cathegories** of GO terms (Molecular Function, Biological Process, Cellullar Component and Human Phenotype), in contrast with the developed model in this thesis, which predicts all GO types together, following the hypothesis that all the GO terms types are related and therefore the features obtained by the process of the optimization of a Neural Network should be useful for all the branches.
In the implemented model based on Convolutional Neural Networks, so as to find the most suitable model, the learning rate was tunned. A random logarithmic exploration was done at the beginning, after the insight that the best

models were found in the learning rate interval $[0.001, 0.01]$. Regarding the architecture of the model, it was explored **up to 3 CNNs** of deepness with **filter sizes 3, 7 and 11** and **number of filters 100, 150 and 200**.
Therefore, aside from the model described in section 3.3, these models were also tested:

Table 3: Deep Learning Model based on Convolutional Neural Networks with 2 Convolutional Layer. The input dimensions are (batchsize, A, L) where A is the alphabet size and L is the sequence lenght that appears in the MSA file. We are forced to use a batchsize of 1 due to the different dimensions of the PSSMs, that depend on the sequence length of each MSA, L, to avoid dimensionality problems. The number of units in the Dense Layer is equal to the number of GO terms taken into account in the analysis. As a result, with the subsets tested that have 4, 17 and all PSSMs respectively, they could be 10, 67 or 12000 neurons.

Input Layer (1,26, L)
Convolutional Layer: filter size = 7, number of filters = 100
Convolutional Layer: filter size = 3, number of filters = 150
Global Pool Layer
Dense Layer: number of units depends on the number of PSSMs used

Table 4: Deep Learning Model based on Convolutional Neural Networks with 3 Convolutional Layer. The input dimensions are (batchsize, A, L) where A is the alphabet size and L is the sequence lenght that appears in the MSA file. We are forced to use a batchsize of 1 due to the different dimensions of the PSSMs, that depend on the sequence length of each MSA, L, to avoid dimensionality problems. The number of units in the Dense Layer is equal to the number of GO terms taken into account in the analysis. As a result, with the subsets tested that have 4, 17 and all PSSMs respectively, they could be 10, 67 or 12000 neurons.

Input Layer (1,26, L)
Convolutional Layer: filter size = 11, number of filters = 100
Convolutional Layer: filter size = 7, number of filters = 150
Convolutional Layer: filter size = 3, number of filters = 200
Global Pool Layer
Dense Layer: number of units depends on the number of PSSMs used

The results of all the architectures are very similar, therefore the simpler model was chosen so as to do the analysis (1 CNN with 100 filters of size 3). The best results are achieved by applying a correction in the **loss** function in order to give more importance to the prediction of 1's in the target vector (real functions of the test protein). The original loss function was **binary crossentropy** but after obtaining this loss we **downweight the 0's** by applying:

$$L = L \cdot (y^{pred} + balance\_coeff)$$

$$L = \frac{L}{\|L\|}$$

where $balance\_coeff$ usually takes values between the interval $[0.0001, 0.001]$ to downweight the 0's. You could see that, for example, using 0.001 as $balance\_coeff$ you are downweighting the loss of 0's by 0.001 while the contribution to the loss of the 1's is almost the same. A **normalization** of the loss is included so as to mantain its values between 0 and 1. Here is shown the same model as in 3.3 by applying $balance\_coeff = 0.0001$:

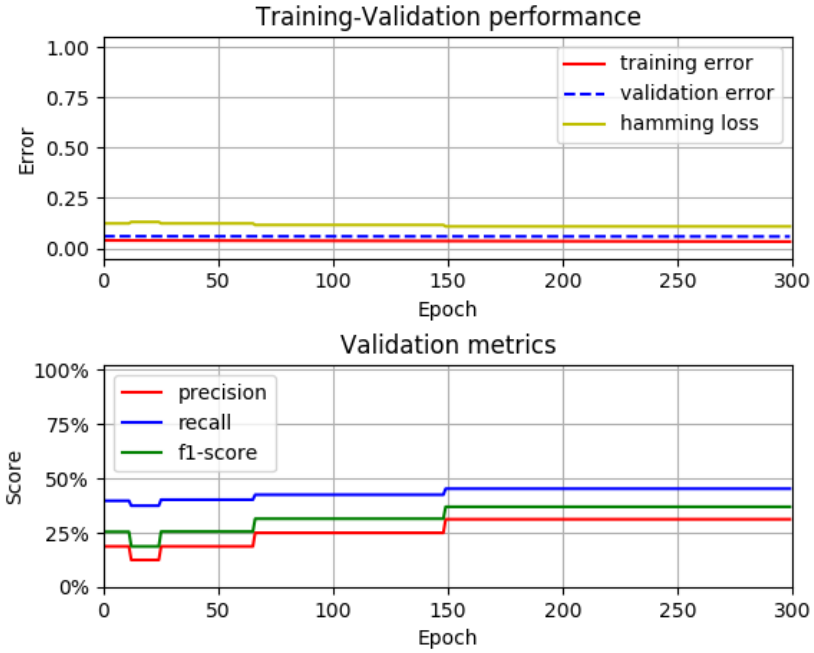


Figure 28: Results extracted from training and validation of the model using 4 PSSMs with $balance\_coeff = 0.0001$. Compared to the model presented in section 3.3, it can be noticed an slightly better performance, achieving an F1 score higher than before, thanks to the implementation and the correct tunning of the balance coefficient.

You may notice that the training error and the validation error start at a very low value; this looks like the model is overfitting the data although the reality is that the model is still predicting a lot of 0's from the begining, achieving a low training, validation and hamming loss.

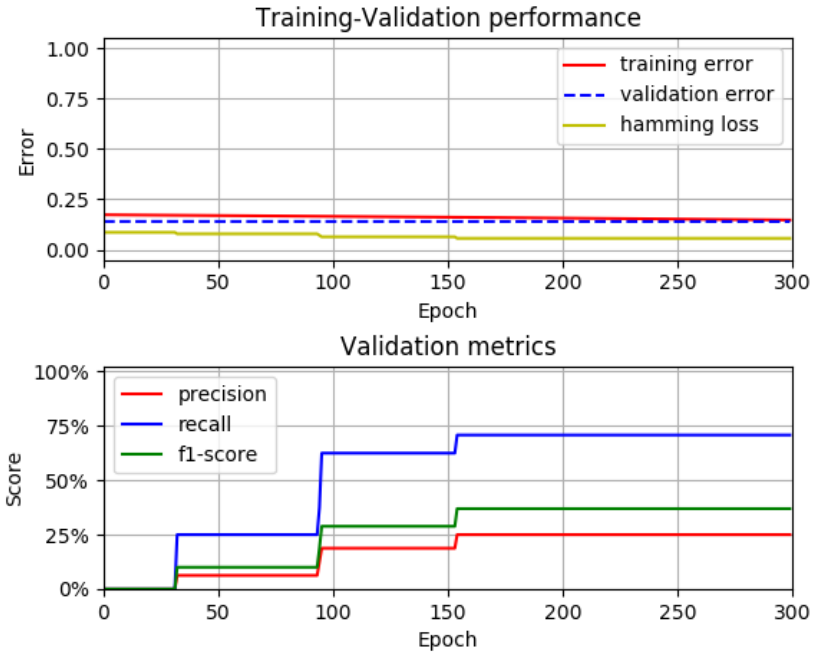


Figure 29: Results extracted from training and validation of the model using 17 PSSMs with $balance\_coeff = 0.0001$. The behaviour of this model is very similar to the one that uses 4 PSSMs.

In order to correctly tune this model, which uses 17 PSSMs, an L_2 norm **regularization** is applied to all the layers of the network. Therefore, the new loss function is:

$$L = L \cdot (y^{pred} + balance\_coeff) + L_2$$

$$L = \frac{L}{\|L\|}$$

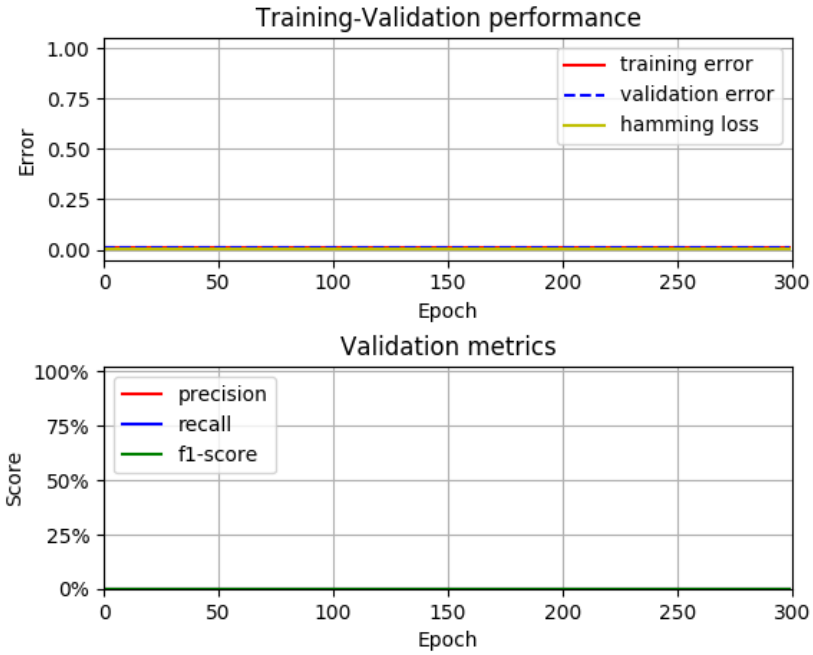


Figure 30: Results extracted from training and validation of the model using all the data available. It is surprisingly that the results are far worse than the ones obtained with 17 PSSMs. This could be caused because the average number of GO terms per protein cluster remains constant, 4 annotations per cluster, while the number of possible protein functions increase a lot, achieving 12000 possible GO terms. Therefore the problem caused by the sparsity in the data becomes impossible to overcome for the model, even using regularization or trying to properly adjust the balance coefficient.

Yoy may notice that even using regularization and the balance coefficients, when all the available data (more than 10000 PSSMs) is used there is no success in the model tuning due to the heavy increase in the sparsity (more than 12000 different GO terms), causing no increase at all in all the loss functions and a constant precision, recall and f1 score of 0. By inspecting the outputs of the model, each GO probability is either really close to 0.5 or really close to 1, depending on the values used for the balance coefficient. To face this problem, data augmentation using **MSA pruning** is proposed. The idea is to build the **phylogenetic tree** of each MSA and then remove some sequences which are located as the leaves of the tree based on its phylogenetic structure. This method introduces a biologically plausible, meaningful and coherent noise.

Figure 31: How to perform pruning using Clustal Muscle

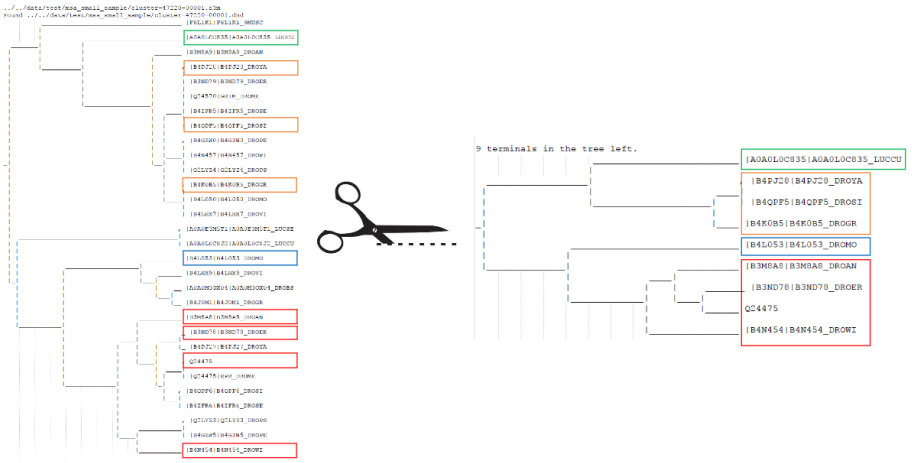


Figure 32: Example of pruning applied to protein sequences

This procedure takes an MSA file and generates different MSAs (in the version implemented, exactly 5),assuming that they **share the same protein functions** and consequently reduces the sparsity of the data. Here there are presented the results after aplying MSA pruning to the 4 PSSMs and 17 PSSMs subsets:

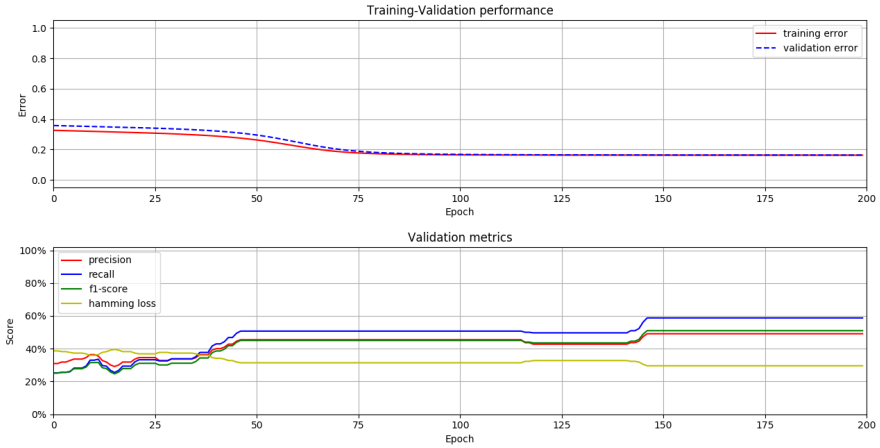


Figure 33: Results extracted from training and validation of the model using MSA pruning on 17 MSAs. In this case, 24 PSSMs are taken into account in the analysis (4 original PSSMs plus 20 produced using MSA pruning, 5 extra per each MSA file). The model behaviour is improved, reaching an F1 score bigger than 0.5. This confirms, as believed, that MSA pruning helps to overcome the problem of heavy sparsity presented in the data.

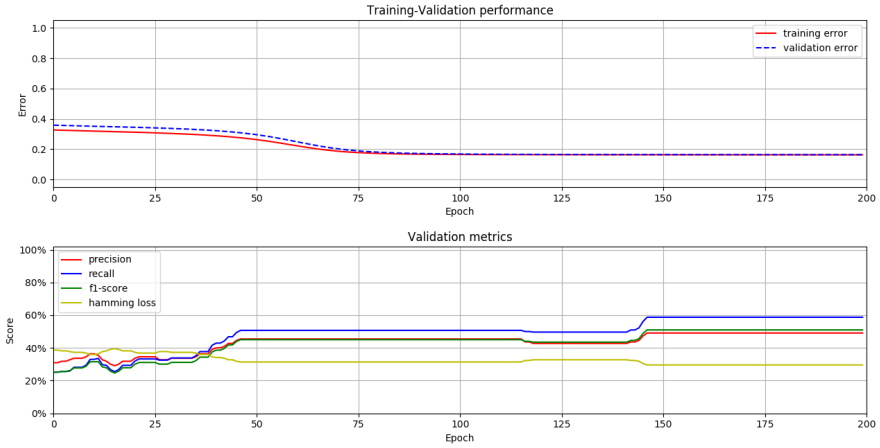


Figure 34: Results extracted from training and validation of the model using MSA pruning on 17 MSAs. In this case, 92 PSSMs are taken into account in the analysis (17 original PSSMs plus 85 produced using MSA pruning, 5 extra per each MSA file). The model behaviour is improved, reaching an F1 score bigger than 0.5. This confirms, as believed, that MSA pruning helps to overcome the problem of heavy sparsity presented in the data.

A bigger f1 score is obtained using MSA pruning but also it could be appreciated that the decrease of the loss functions and the increase of precision, recall and f1 score becomes more natural.
Unfortunately, even though MSA pruning could be computed on the go, due to the bottleneck of calculating the PSSMs for the new MSAs, this data augmentation method could not be applied to the whole dataset.

6 Conclusions

Some evaluations methods such as **Hamming loss** were just useful in order to detect why the model was not working as it was supposed to. For this kind of data, where a protein usually has just 5 functions out of 5700 possible anotations it was easy for the model to just claim that a protein has no function at all so as to get a low hamming loss. To check the good performance of the model it was better to pay attention to other evaluation methods such as **precision, recall and f1 score**.

The data is too **sparse** and therefore is complicated to work properly with it using our models. There are similar situations in the field of Machine Learning which present almost the same problems, such as **fraud detection**. In these challenging problems there is also a huge umbalance in the data, but it can be faced via **data augmentation** and **data reduction**. There is just a matter of reducing the non-fraud samples and increasing the fraud samples in a proper way. This is much more difficult to do with our dataset, because in order to correctly balance the data there should be "invented" new proteins with lots of functions, to face the sparseness, and because we are meeting a **multilabel** classification problem instead of just a normal classification problem. Then the use of **MSA pruning** as a data augmentation method could really help with these circumstances. Another thing to be considered is the possibility of just pay attention to a **subset of GO terms**, simplifying the problem by just being able to predict GOs from this subset.

As a final conclusion, the use of CNNs and PSSMs for predicting the protein functions is a great insight but the **need of more information/data** so as to do this task is obvious.

7 Future Work

The following future work is proposed, listed below.

7.1 Data Preprocessing on the go

Preprocessing the data and building the PSSMs for the different MSAs is a time-consuming task and requires powerful computers and furthermore disk space so as to store the data. A great enhancement could be the realization of this part of the process **on the go**, meaning that no PSSMs storage is needed (these are calculated when necessary). It should be considered the implementation of some **web services** that perform these tasks to speed up the process, using precomputed data, such as finding to which MSA a protein belong and PSSMs.

7.2 Deal with the data sparsity

The data sparsity leads the algorithms to the predictions of all 0's in the target vector, due to the imbalance presented in the data. Although it has been tried to overcome this problem, it is still to be tested how the **MSA pruning** will affect the algorithm in the analysis of the full dataset and if the use of different **loss functions** could help.

7.3 Consider more data types as input

There are still a lot of approaches that could improve the prediction of the protein functions. During the development of this thesis, it has been considered the development of a **Big Network**, which uses several **branches** forming a big architecture.

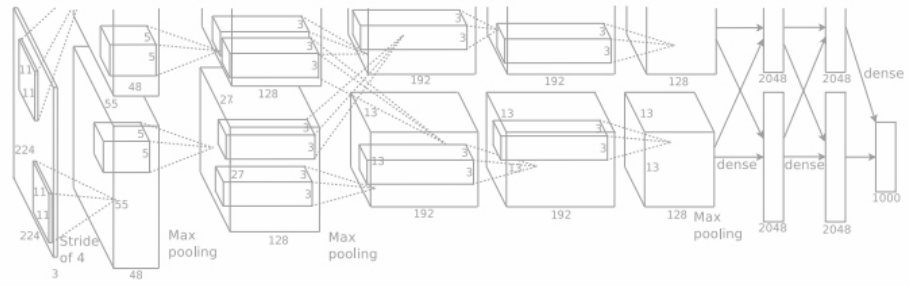


Figure 35: KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. Imagenet classification with deep convolutional neural networks. En Advances in neural information processing systems. 2012. p. 1097-1105. Example of a Deep Neural Network with more than one branch.

Therefore, the use of the protein clusters and the respectively generation of a CNN using PSSMs as inputs could be just some **features** to take into account in order to predict protein functions. It should be considered the use of other protein-based data such as raw **protein sequence** or the proteins pairwise comparison of **sequence difference** as input so as to improve the predictions.

7.4 Consider relations between labels

In the model created, GO terms are considered as independent classification categories, while they are **graph structured** and they have strongly relations between them. Trying to build a GO graph using even high-level GO terms that are not taking into account in this analysis because they are too general. For example, nucleic acid binding is not considered as a GO term in the dataset used while DNA binding is. Detecting that a **higher level annotation** is presented in a protein could help to predict other annotations, such as the nucleic acid binding term will help to predict DNA binding or RNA binding terms. This information could be exploited to obtain more accurate predictions.

7.5 Fair comparison with other CAFA2 methods

In order to fairly compare the model presented in this thesis with the other CAFA2 methods, it must be developed a layer/system that works before the input layer of the CNN model that associate a protein with its correspondent protein cluster, that will be used as input for this model. Up to date, the CNN model has only been tested by directly inputing protein clusters to it.
In addition, the model developed in this project tries to predict all the cathegories of protein function (Cellular Component, Biological Process and Molecular Function) at once, while CAFA2 methods focused on one type of ontology for doing the predictions.

References

[1] JIANG, Yuxiang, et al. An expanded evaluation of protein function prediction methods shows an improvement in accuracy. Genome biology, 2016, vol. 17, no 1, p. 184.

[2] CAFA3, `http://biofunctionprediction.org/cafa/`. Accessed: 27-09-2017

[3] JANIN, Jol. Assessing predictions of proteinprotein interaction: the CAPRI experiment. Protein science, 2005, vol. 14, no 2, p. 278-283.

[4] ALTSCHUL, Stephen F., et al. Basic local alignment search tool. Journal of molecular biology, 1990, vol. 215, no 3, p. 403-410.

[5] ZHANG, Chengxin; FREDDOLINO, Peter L.; ZHANG, Yang. COFACTOR: improved protein function prediction by combining structure, sequence and proteinprotein interaction information. Nucleic Acids Research, 2017.

[6] GONG, Qingtian; NING, Wei; TIAN, Weidong. GoFDR: a sequence alignment based method for predicting protein functions. Methods, 2016, vol. 93, p. 3-14.

[7] YOU, Ronghui, et al. GOLabeler: Improving Sequence-based Large-scale Protein Function Prediction by Learning to Rank. bioRxiv, 2017, p. 145763.

[8] GOLKOV, Vladimir, et al. 3D Deep Learning for Biological Function Prediction from Physical Fields. arXiv preprint arXiv:1704.04039, 2017.

[9] GOLKOV, Vladimir, et al. q-Space deep learning: twelve-fold shorter and model-free diffusion MRI scans. IEEE transactions on medical imaging, 2016, vol. 35, no 5, p. 1344-1351.

[10] KARPATHY, Andrej, Neural Networks. `http://cs231n.github.io/neural-networks-1/`. Accessed: 27-09-2017.

[11] KARPATHY, Andrej, Convolutional Neural Networks. `http://cs231n.github.io/convolutional-networks/`. Accessed: 27-09-2017.

[12] KARPATHY, Andrej, The process of learning. `http://cs231n.github.io/neural-networks-3/`. Accessed: 27-09-2017.

[13] JUNCKER, Agnieszka S., et al. Sequence-based feature prediction and annotation of proteins. Genome biology, 2009, vol. 10, no 2, p. 206.

[14] DAS, Sayoni; ORENGO, Christine A. Protein function annotation using protein domain family resources. Methods, 2016, vol. 93, p. 24-34.

[15] PRECHELT, Lutz. Early stopping-but when?. Neural Networks: Tricks of the trade, 1998, p. 553-553.

[16] LI, Li; WANG, Houfeng. Towards Label Imbalance in Multi-label Classification with Many Labels. arXiv preprint arXiv:1604.01304, 2016.

[17] KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. Imagenet classification with deep convolutional neural networks. En Advances in neural information processing systems. 2012. p. 1097-1105.

[18] BISHOP, Christopher M. Pattern recognition and machine learning. springer, 2006.

[19] BISHOP, Christopher M. Neural networks for pattern recognition. Oxford university press, 1995.

[20] RADIVOJAC, Predrag. A (not so) quick introduction to protein function prediction. 2013.

[21] KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. Imagenet classification with deep convolutional neural networks. En Advances in neural information processing systems. 2012. p. 1097-1105.

[22] COCK, Peter JA, et al. Biopython: freely available Python tools for computational molecular biology and bioinformatics. Bioinformatics, 2009, vol. 25, no 11, p. 1422-1423.

[23] BERGSTRA, James, et al. Theano: Deep learning on gpus with python. 2011.

[24] UNIPROT CONSORTIUM, et al. UniProt: a hub for protein information. Nucleic acids research, 2014, p. gku989.

[25] GENE ONTOLOGY CONSORTIUM, et al. The Gene Ontology (GO) database and informatics resource. Nucleic acids research, 2004, vol. 32, no suppl 1, p. D258-D261.

[26] CARBON, Seth, et al. AmiGO: online access to ontology and annotation data. Bioinformatics, 2008, vol. 25, no 2, p. 288-289.

[27] CIFAR-10, `https://www.cs.toronto.edu/~kriz/cifar.html`. Accessed: 27-09-2017.